Gaussian Processes and Kernels

In this note we'll look at the link between Gaussian processes and Bayesian linear regression, and how to choose the kernel function.

Bayesian linear regression as a GP

The Bayesian linear regression model of a function, covered earlier in the course, is a Gaussian process. If you draw a random weight vector $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \sigma_w^2 I)$ and bias $b \sim \mathcal{N}(0, \sigma_b^2)$ from Gaussian distributions[1], the joint distribution of any set of function values, each given by

$$f(\mathbf{x}^{(i)}) = \mathbf{w}^\top \mathbf{x}^{(i)} + b,$$

is Gaussian. If the distributions over the weights are prior beliefs, then we are specifying a Gaussian process prior on a function.

A Gaussian Process created by a Bayesian linear regression model is *degenerate* (boring), because the function has to be linear in $\mathbf{x}$. Once we know the function at $(D+1)$ input locations (in general position), we can solve for the weights, and we know the function everywhere. If we use K basis functions, the function is constrained to be a linear combination of those basis functions. Knowing the function at only K positions is enough to find the basis functions' coefficients and know the function everywhere.

By choosing an appropriate kernel function, we can define Gaussian processes that are more interesting. The covariance matrices we produce can be full rank, which would mean that no matter how many function values we see, there would still be more to learn about the function.

Bayesian linear regression's kernel

As Bayesian linear regression defines a Gaussian process, there is a corresponding kernel function that specifies all of the covariances in the model:

$$\begin{aligned} f(\mathbf{x}^{(i)}) &= \mathbf{w}^\top \mathbf{x}^{(i)} + b, \qquad \mathbf{w} \sim \mathcal{N}(\mathbf{0}, \sigma_w^2 \mathbb{I}), \;\; b \sim \mathcal{N}(0, \sigma_b^2) \\ \operatorname{cov}(f_i, f_j) &= \mathbb{E}[f_i f_j] - \overbrace{\mathbb{E}[f_i]\mathbb{E}[f_j]}^{0} \\ &= \mathbb{E}\left[(\mathbf{w}^\top \mathbf{x}^{(i)} + b)^\top (\mathbf{w}^\top \mathbf{x}^{(j)} + b)\right] \\ &= \sigma_w^2 \mathbf{x}^{(i)\top} \mathbf{x}^{(j)} + \sigma_b^2 \;=\; k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}). \end{aligned}$$

The kernel is specified by parameters σ_w^2 and σ_b^2, which are *hyperparameters* in a Bayesian hierarchical model — parameters that specify the prior on parameters.

It's usually more efficient to implement Bayesian linear regression directly, than interpreting it as a Gaussian process. Although if there are more input features or basis functions than data-points, using the Gaussian process view is cheaper.

Feature spaces and the kernel trick

We can replace the original features $\mathbf{x}$ with a vector of basis function evaluations $\boldsymbol{\phi}(\mathbf{x})$ in the linear regression model. We obtain the kernel:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \sigma_w^2 \boldsymbol{\phi}(\mathbf{x}^{(i)})^\top \boldsymbol{\phi}(\mathbf{x}^{(j)}) + \sigma_b^2.$$

To get arbitrarily complicated functions, we would need an infinite $\boldsymbol{\phi}$ vector of features. Fortunately, all we need to know are inner products $\boldsymbol{\phi}(\mathbf{x}^{(i)})^\top \boldsymbol{\phi}(\mathbf{x}^{(j)})$ — the feature vector never occurs by itself in the maths. If we can get the inner products directly, without creating

1. Yes, I've just changed notation *again*; sorry. The prior on weights has also been labelled with variance $1/\alpha$ and σ^2_{prior}. I'll unify this notation for next year.

the infinite feature vectors, we can infer an infinitely complicated model, with a finite amount of computation.

Replacing an inner product of features with a simple kernel function, corresponding to a large or infinite set of basis functions, is known as the *kernel trick*. Some kernels are derived explicitly as inner products of an infinite collection of basis functions. Moreover, it can be shown that Mercer kernels, or positive definite kernels, all correspond to taking inner products in some feature space, which might be infinite.

Several other machine learning algorithms can be expressed in terms of only inner products, and so can be *kernelized*. For example PCA can be written using only inner products. Most famously, a linear classifier known as the Support Vector Machine (SVM) is usually used in its kernelized version. Gaussian processes are kernelized Bayesian linear regression.

Choosing a family of kernel functions

We can't just make up *any* function for $k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$. We have to know that it will always give positive semi-definite matrices (and usually we want positive definite matrices), so that the prior on functions is actually a valid Gaussian distribution.

The textbooks have various options, and we won't go through them all. I'll briefly highlight some of the things that are worth knowing.

Going beyond feature vectors: There are kernel functions that can compare two strings or graphs, and return a covariance. Kernel methods give a flexible means to model functions of structured objects.

Kernels can be combined in various ways. For example, given two positive definite kernel functions, a positive combination:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \alpha k_1(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) + \beta k_2(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}), \quad \alpha > 0,\ \beta > 0,$$

is also positive definite. (Can you prove it?)

Kernels can give functions with qualitatively different properties: The examples in these notes come from Gaussian kernels. The functions from this prior are ridiculously smooth for many purposes, and other choices may be better. (In high-dimensions you can't really see any detail of a function, and the smoothness of the Gaussian kernel probably matters less.)

Kernels usually have parameters. Most kernels have free parameters that change the distribution over functions. The combination of kernels above introduced two extra parameters α and β. As explained above, the link to Bayesian linear regression means that these parameters are often called hyperparameters. Tuning hyperparameters is the main way that we control the prior of a Gaussian process. The choice of prior strongly affects the model's generalization performance for a given amount of data.

Hyperparameters for the Gaussian kernel

The Gaussian kernel can be derived from a Bayesian linear regression model with an infinite number of radial-basis functions. You might see several other names for the kernel, including RBF, squared-exponential, and exponentiated-quadratic.

The version below has hyperparameters σ_f, ℓ_d to control its properties,

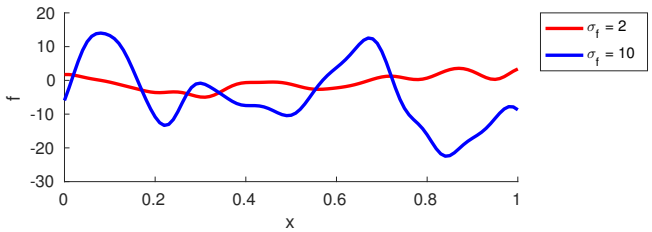
$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \;=\; \sigma_f^2 \exp\Bigg(-\tfrac{1}{2} \sum_{d=1}^{D} (x_d^{(i)} - x_d^{(j)})^2 / \ell_d^2 \Bigg).$$

Other kernels have similar parameters, but we will demonstrate their effect just on the Gaussian kernel.

The marginal variance of one function value f_i is:

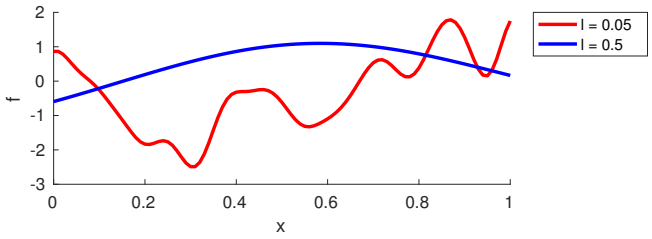
$$\text{var}[f_i] = k(\mathbf{x}^{(i)}, \mathbf{x}^{(i)}) = \sigma_f^2.$$

The figure below illustrates functions drawn from priors with different marginal variance (also called "function variance" or "signal variance") σ_f^2:



If we saw a long enough stretch of function, then $\approx 2/3$ of the points would be within $\pm\sigma_f$ of the mean (which is zero in all the figures here). Although we often won't see really long stretches of a function.

The number of turning points in the function is random, but controlled by the lengthscale ℓ. (Some papers refer instead to a "bandwidth" h, as was traditionally used with RBF models, but is less used with GPs.) As in the equation above, each feature dimension might have its own lengthscale ℓ_d, or there might be a single shared lengthscale ℓ. The plot below illustrates samples with two different lengthscales:



The lengthscale indicates the typical distance between turning points in the function. Often a good value is comparable to the range of a feature in the training data, because functions in many regression problems don't have many turning points.

Learning the hyperparameters

If we have Gaussian observations of the function $\mathbf{y}$, the probability of the data is just a Gaussian. The log of that probability is the log of the standard multivariate Gaussian pdf:

$$\log p(\mathbf{y} \mid X, \theta) = -\tfrac{1}{2}\mathbf{y}^\top M^{-1}\mathbf{y} - \tfrac{1}{2}\log|M| - \tfrac{N}{2}\log 2\pi,$$

where $M = K + \sigma_n^2 I$, is the kernel matrix evaluated at the training inputs plus the variance of the observation noise. The model parameters θ are the kernel parameters $\{\ell_d, \sigma_f, \dots\}$, which control K, and the noise variance σ_n^2.

One way to set the parameters is by maximum likelihood: find values that make the observations seem probable. Gradients of the log-likelihood with respect to the hyperparameters can be computed, so we can use gradient-based optimizers. Because the GP can be viewed as having an infinite number of weight parameters that have been integrated out, $\log p(\mathbf{y} \mid X, \theta)$ is often called the *marginal* likelihood.

Optimizing the marginal likelihood of a stochastic process *sounds* fancy. But all we are doing is fitting a Gaussian distribution (albeit a big one) to some data, and the *marginal likelihood* is just a standard Gaussian pdf. A minimal implementation isn't much code.

It is quite possible to overfit by fitting the hyperparameters. Regularizing the log noise variance $\log \sigma_n^2$, preventing it from getting too small, is often a good idea.

A fully Bayesian approach is another way to avoid overfitting. It computes the posterior over a function value by integrating over all possible hyperparameters:

$$p(f_* \mid \mathbf{y}, X) = \int p(f_* \mid \mathbf{y}, X, \theta)\, p(\theta \mid \mathbf{y}, X)\, \mathrm{d}\theta$$

However, this integral can't be computed exactly. The first term in the integrand is tractable. The second term is the posterior over hyperparameters, which comes from Bayes' rule, but requires approximation before we can compute the integral.

Computation cost and limitations

Gaussian processes show that we can build remarkably flexible models and track uncertainty, with just the humble Gaussian distribution. In machine learning they are mainly used for modelling expensive functions. But they are also used in a large variety of applications in other fields (sometimes under a different name).

The main downside to straightforward Gaussian process models is that they scale poorly with large datasets.

- The $O(N^3)$ computation cost usually takes the blame, required to factor the M matrix above so that we can evaluate the marginal likelihood and make predictions.
- That cost isn't the only story. Computing the kernel matrix costs $O(DN^2)$ and uses $O(N^2)$ memory. Sometimes the covariance computations can be significant, and running out of memory places a hard limit on problem sizes.
- There is a large literature on special cases and approximations of GPs that can be evaluated more cheaply.

Not all functions can be represented by a Gaussian process. The probability of a function being monotonic under any Gaussian process is zero.

Gaussian processes are a useful building block in other models. But as soon as our observation process isn't Gaussian, we have to do rather more work to perform inference, and we need to make approximations.

Check your understanding

Here are questions about the lengthscale parameter(s) ℓ_d in the squared exponential kernel. The answers aren't explicitly in the notes, you'll have to think about what the prior model says, and so reason about what will be imposed on the posterior.

- If a GP uses a kernel with a lengthscale that is too short, what will happen and why? You don't need to do maths: sketch what draws from the prior would be like, and use your intuition to see what posterior samples would look like given a few datapoints.
- What will happen as the lengthscale ℓ is driven to infinity?
- If a kernel has a different lengthscale ℓ_d for each feature-vector dimension, what happens if we drive one of these lengthscales to infinity?

Further Reading

The readings from the previous note still stand.

Carl Rasmussen has a nice example of how GPs can combine multiple kernel functions to model interesting functions:
`http://learning.eng.cam.ac.uk/carl/mauna/`

Some Gaussian process software packages:

- `https://github.com/GPflow/GPflow`
  Uses automatic differentiation and TensorFlow but new and in development.
- `http://sheffieldml.github.io/GPy/`
  More mature Python codebase.

- `http://www.gaussianprocess.org/gpml/code/matlab/doc/`
  Nicely structured Matlab code base. (Used in Carl's demo above.)
- `http://becs.aalto.fi/en/research/bayes/gpstuff/`
  Matlab. Less easy-to-follow code, but implements some interesting things.
- `https://github.com/HIPS/Spearmint`
  Bayesian optimization. Watch out for the license.

Practical tips

As with any machine learning method, the machines can't do everything for us yet. Visualize your data. Look for weird artifacts that might not be well captured by the model. Consider how to encode inputs (one-hot encoding, log-transforms, ...) and outputs (log-transform? ...).

To set initial hyper-parameters, use domain knowledge wherever possible. Otherwise...

- Standardize input data and set (initial) lengthscales ℓ_d to ~ 1.
- Standardize targets and set the function variance σ_f^2 to ~ 1.
- Sometimes useful: set the initial noise level high, even if you think your data have low noise. The optimization surface for your other parameters will be easier to move in.

If optimizing hyper-parameters, (as always) random restarts or other tricks to avoid local optima may be necessary.

Don't believe any suggestions that maximizing marginal likelihood can't overfit. It can, so watch out for it, and consider more Bayesian alternatives.